Linguagens de Domínio Específico

Fabio Mascarenhas – 2017.1

http://www.dcc.ufrj.br/~fabiom/dsl

Combinadores *scannerless*

- Uma vez que adicionamos predicados sintáticos, podemos fazer nossos combinadores atuarem diretamente em cima do texto, e não de uma sequência de tokens

- Precisamos apenas de combinadores especiais para fornecer informação suficiente sobre tokens da entrada para nossas mensagens de erro baseadas nas falhas mais distantes

- Com um pouco de abstração nossa *DSL* para escrever parsers não fica muito diferente da gramática léxica+sintática original

PEGs

- As gramáticas de expressões de parsing, ou PEGs (parsing expression grammars) são uma linguagem para especificar parsers determinísticos

- Ao contrário das gramáticas livres de contexto, PEGs têm um mapeamento natural para os combinadores que estamos usando

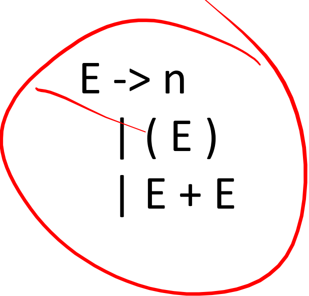
- Uma *expressão de parsing* pode ser a expressão vazia `''`, um *terminal* `'a'`, um *não-terminal* `A`, uma *sequência* `pq`, onde `p` e `q` são expressões de parsing, uma *escolha ordenada* `p/q`, uma *repetição* `p*`, ou um *predicado* `!p`

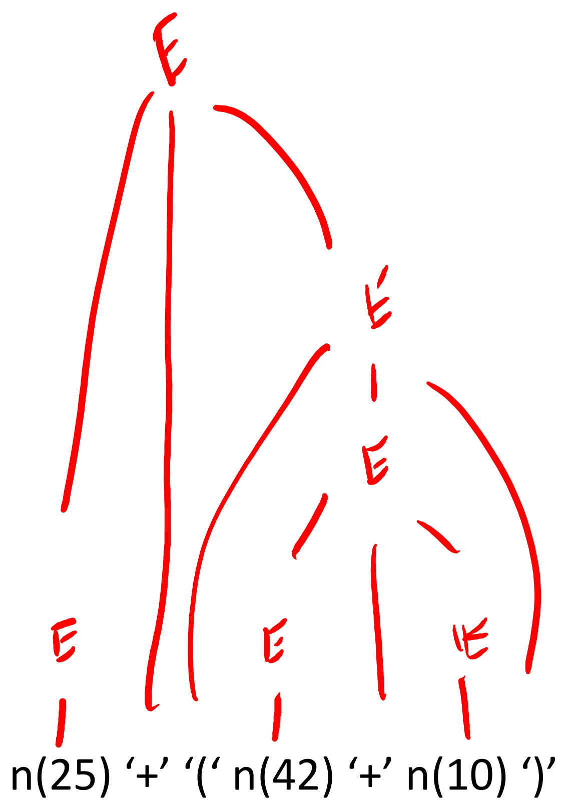
- Uma PEG é simplesmente um mapeamento entre nomes de não-terminais e suas expressões de parsing

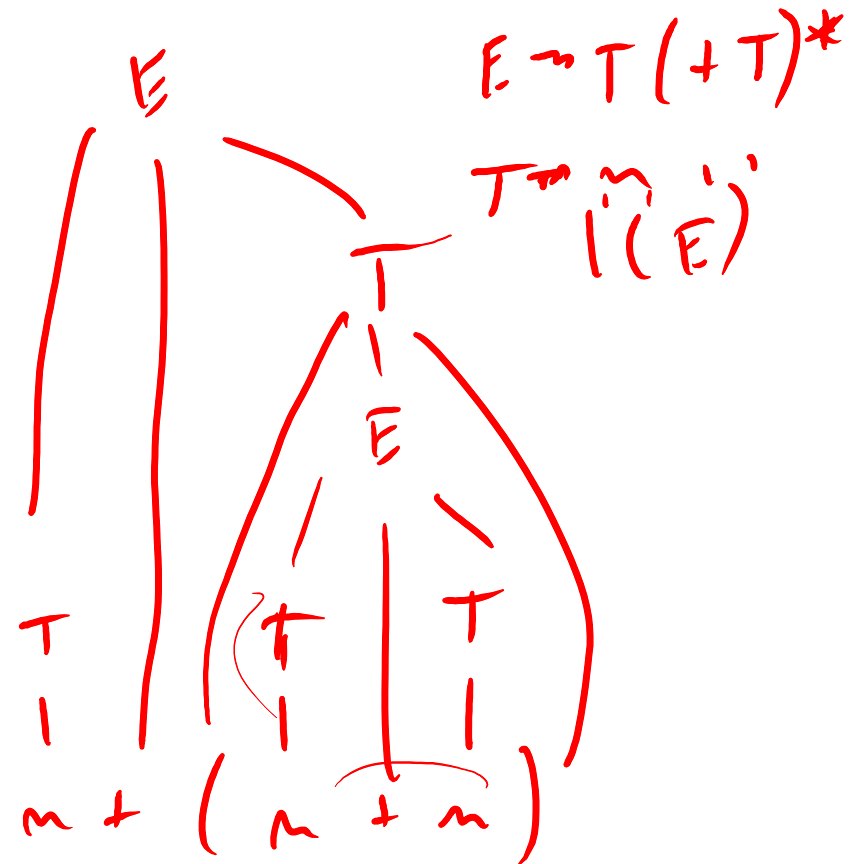
Árvore Sintática e AST

- A estrutura gramatical de uma entrada forma naturalmente uma árvore: tokens são folhas, outros termos sintáticos são nós internos
- Mas esse tipo de árvore tem muita informação redundante, e está intimamente ligada à gramática
- Nem sempre a gramática mais natural para uma linguagem é a melhor para escrever um analisador para ela
- Resolvemos os dois problemas com uma *árvore sintática abstrata*: toda informação redundante é descartada, e procuramos a representação ideal da estrutura gramatical daquela linguagem, independentemente da gramática concreta que usamos

Exemplo – árvore sintática





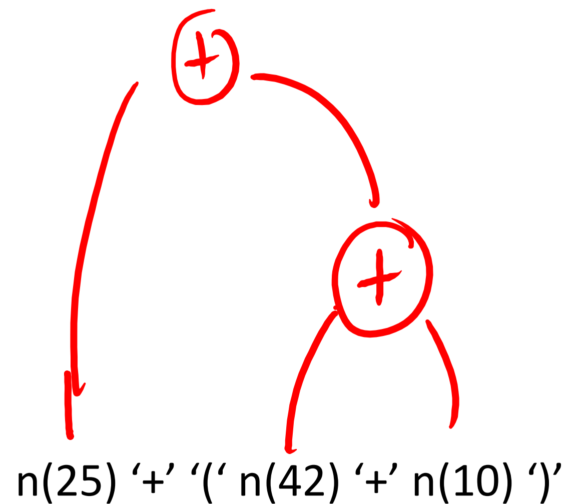


Exemplo - AST

E -> n
  | (E)
  | E + E

E -> T (+ T)*

T -> n
  | '(' E ')'



Características de uma AST

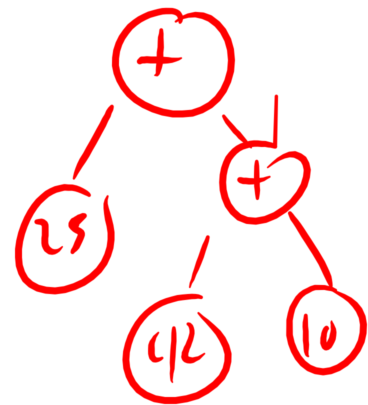
- *Densa*: não há nós redundantes
- *Conveniente*: é fácil de percorrer e processar
- *Significativa*: Enfatiza operadores, operandos, e a relação entre eles, e não artefatos da gramática
- Podemos precisar escrever múltiplas passadas no processo de análise de um programa e a construção de seu modelo semântico
- Mudanças na gramática devido à conveniência da análise sintática não devem afetar a AST

ASTs heterogêneas

- A forma mais comum de implementar uma AST é usar uma classe diferente para cada tipo de nó que temos

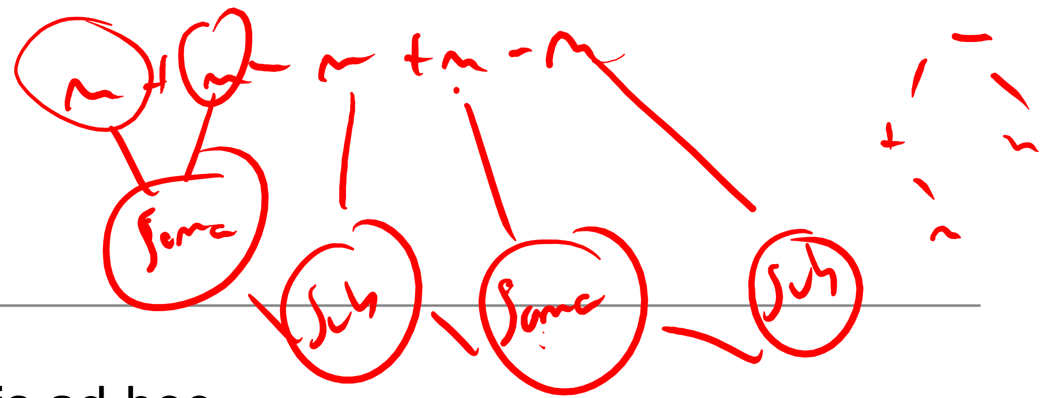
- Classes abstratas ou interfaces implementam estruturas sintáticas que possuem diversos tipos concretos de nó

```
public abstract class AST {
    Token token;
    // missing normalized list of children; subclasses define fields
}
public abstract class ExprNode extends AST { ... }
public class AddNode extends AST {
    ExprNode left, right;          // irregular, named fields
    «fields-specific-to-AddNode»
}
```



- Se estamos escrevendo o código para criar e processar as árvores manualmente então essa forma é a ideal

Construindo ASTs (1)



- Construir uma AST heterogênea é mais ad-hoc
- Em um analisador recursivo, cada regra pode retornar o pedaço da AST que ela representa
- O corpo de cada regra compõe os pedaços do jeito que quiser

```
public Exp exp() {
  Exp res = termo();
  while(la.tipo == '+' || la.tipo == '-') {
    if(la.tipo == '+') {
      match('+');
      res = new Soma(res, termo());
    } else {
      match('-');
      res = new Sub(res, termo());
    }
  }
  return res;
}
```

```
public List<Event> events() {
  ArrayList<Event> res =
    new ArrayList<>();
  match(StateMachineLexer.EVENTS);
  do {
    res.add(event());
  } while(la.tipo ==
          StateMachineLexer.NAME);
  match(StateMachineLexer.END);
  return res;
}
```

Construindo ASTs (2)

- Com combinadores, precisamos fazer os combinadores poderem retornar um pedaço da AST, além do restante da entrada

- Com tipos genéricos podemos manter combinadores bem genéricos, mas que constroem ASTs heterogêneas

```
public interface Parser<A,B> {
  Result<A,B> parse(List<B> in);
}
```

```
public class Result<A,B> {
  public final A res;
  public final List<B> out;
  ...
}
```

- O combinador de sequência fica mais complicado: precisamos passar cada sequência através de uma função responsável por combinar os pedaços da sequência